Feature Testing on Image Classification for ImageNet

Timothy Chan
Stanford University
timchan@stanford.edu

Abstract

The purpose of this research study is to classify images from TinyImageNet, which contains a subsection of ImageNet classes and a standardized image set for image classification. I created a Neural Network using some interesting feature sets, most prominently SIFT and texture features. While my SIFT features were calculated mainly using code from OpenCV [4], an external source, I conducted several experiments with texture features. I based these texture features off of the work done by R. Haralick [2], but as his were for classifying terrain, and my goal was to classify everyday objects, I modified and experimented with the algorithm. This experiment included two-hundred different training, validating, and testing classes from TinyImageNet to classify the images. In order to determine the accuracy of the varying approaches, training set examples were categorized into folders, which were eventually scored. Although the research was successfully able to implement texture and SIFT features, the results appeared to be unsuccessful in showing much gain in classification accuracy. This loss of gain may be mainly due to unsuccessful tuning of the algorithm.

1. Introduction

Image classification analyzes the pixel properties of various images and organizes these images into categories based off of the similarities between the graphics. It is one of the most important subsections of digital image analysis as it provides meaning behind each visual by analyzing and interpreting the hue, saturation, and brightness of each pixel within the image. When measuring an images composition, users refer use pixels, which are the smallest property of data represented within pictures. This research study centers on classifying images from TinyImageNet, which is a subsection of ImageNet classes that contains a standardized image set for image classification. Image Classification takes an imperative role in navigating the internet and, if accurate enough, will be an important part of allowing computers to have a more direct interaction with the real world. It is because of this that a large amount of highly esteemed organizations work toward making improvements in image classification. Some advanced techniques by companies such as Microsoft and Google have already surpassed humans when it comes to classifying images. When compared with a humans computational potential, these techniques are far more efficient and reliable. Seeing as how image classification plays a significant part in todays society, my desire is to learn more about how image classification works and how it can be applied in many contexts. When classifying an image into a certain category, it is assigned a score that represents how similar it is to another visual. There are many variables and techniques that can make a system score better, such as techniques used to tuning parameters and features, which I find to be the most interesting. As a result, I plan to experiment with several features in order to classify the images. There were two features in particular that I found quite intriguing. My primary focus and goal for this project was to learn and explore these approaches, primarily working on sift descriptors and texture features.

2. Background/Related Work

2.1. SIFT descriptors

A famous and influential feature that is commonly used is SIFT features. I learned about these features from the paper"Distinctive Image Features from Scale-Invariant Keypoints" by Lowe, D [3], as well as another paper that extends SIFT titled "Local Pyramidal Descriptors for Image Recognition," by Seidenari, L [1]. SIFT features as described in Lowe's paper [3] are based on creating a histogram of image gradients in order to see how often specific image gradients appear, as shown in Figure 1. As it uses a histogram, this feature is invariant to changes in illumination. Also, as the features are first rotated by the strongest gradient, SIFT features are also rotation invariant.

In addition to these calculations, an improvement was made by Seidenari [1], in which SIFT features are calculated for different scalings before being compared, as shown in Figure 2. By comparing the SIFT features in this sort of

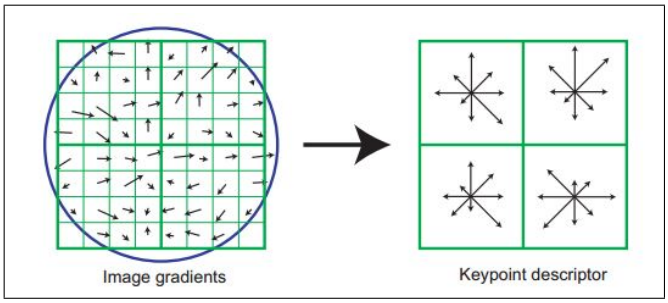

Figure 1. SIFT histogram of gradients from Lowe,D [3]

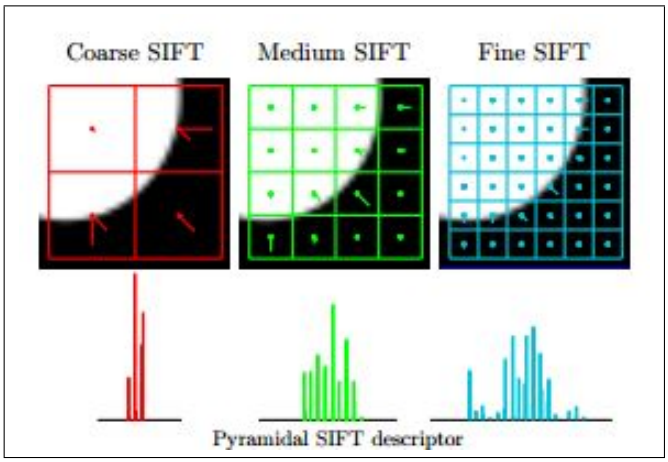

Figure 2. Pyramidal SIFT features from Seidenari, L [1]

pyramid, the feature set can also be scale invariant, making it very strong in recognizing the same object in a wide variety of different situations.

Due to SIFT's prevalent use and usefulness at detecting the same object, I decided to try and apply SIFT to TinyImageNet. While SIFT is better at re-detecting the same feature from different angles, such as following an object on video, I was experimenting to see how it fares in detecting features that are similar but not the same object. For example, how well it can detect different sunglasses on different people's faces, when sunglasses are relatively similar objects, even when they are not the same pair of sunglasses.

2.2. Image Texture

Another set of features is based on texture. I referenced a paper called "Textural Features for Image Classification," by R. Haralick [2], which attempts to quantify how texture would be conveyed in a digital image. While Haralick's paper was mainly a technique to classify landscapes, as seen in Figure 3, it seems like it could be useful for object image recognition as well. Haralick's texture features were created by calculating the contrast and correlation for regions at various degrees around a central region. While this is good for

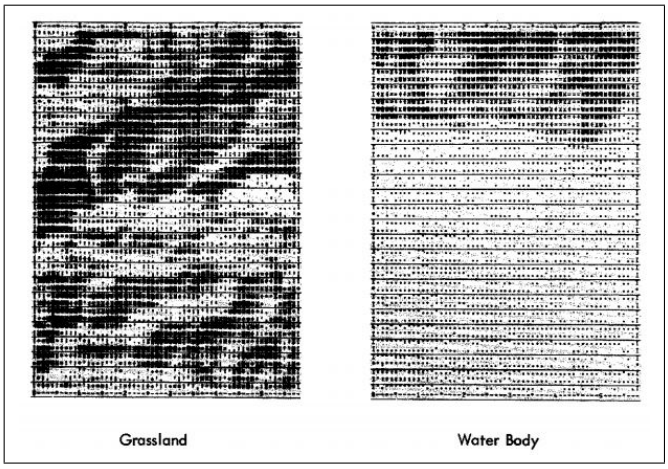

Figure 3. Classes from Haralick, L [2]

classifying textures on a relatively flat background, such as landscape on the ground, I found that it was a bit too sensitive to texture changes that were not quite so repetitive, such as scales wrapping around a fish.

3. Approach

I am planning to test these features on a image classification challenge provided by CS 231N at Stanford. This project is called Tiny ImageNet and will provide a simple way to access data and to see results. I primarily extended the code from our assignments. Because of this, the features that I used came from four categories. The original image pixels, the pretrained models on TinyImageNet-A, SIFT features, and texture features.

The SIFT features came from OpenCV, which is an open source python library for image recognition that has functions that calculate SIFT keypoints as well as does the histogram of gradients calculations to calculate SIFT features. OpenCV has many parameters for SIFT that I used to attempt to tune the model, such as the number of features, the number of layers, and the contrast and edge thresholds.

The texture features I programmed in python based off of the paper by Haralick [2]. I tried several different techniques in attempting to find useful texture features.

For one feature attempt, I did a correlation with areas of input images in relation to nearby areas. Where the correlation with neighboring areas was highest, I would average the area with the surrounding area, and return that as the primary "texture" for the image. One issue faced with this feature selection was that the top texture was generally just a monochrome background texture, such as blue sky or a white wall. I attempted to correct for this by selecting the top several textures, in which whenever a top texture was selected, the correlation score between the top texture and every other texture was subtracted from every other texture's

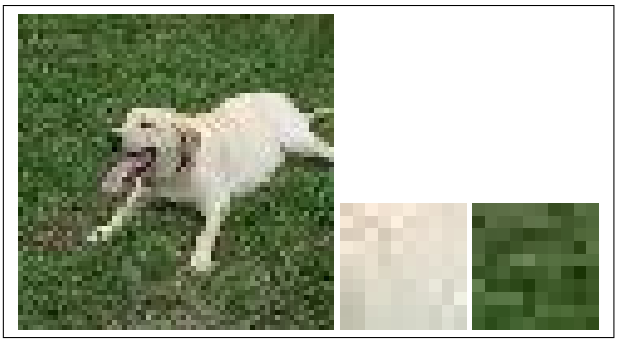

Figure 4. Texture features from an image

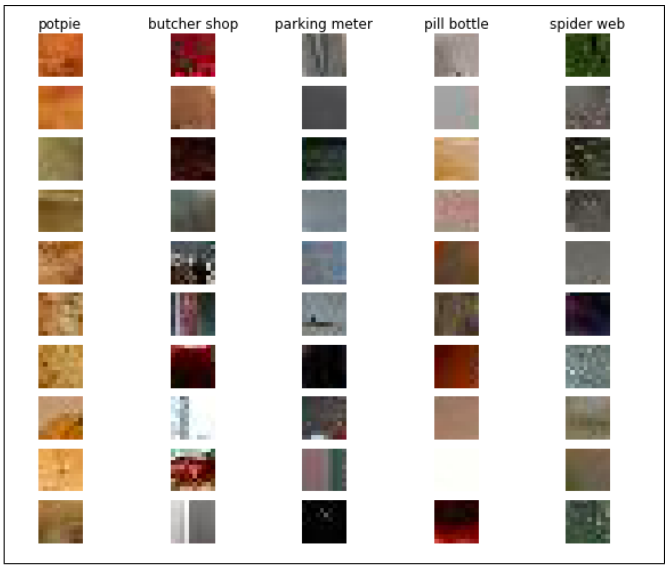

Figure 5. Textures representative of their class

score, as seen in Figure 4. This made it so that different textures would be selected for the next highest texture. While this did return interesting textures based on the image, it did not seem to be able to help the scoring metric.

Another approach I attempted was to find textures that represented a class, and would try to look for these textures to classify the image. The way I did this was that I would pick a random patch, call it a texture, and find its lowest L2 distance from every possible patch in a different image of the same class. After several iterations to try and find better patches, selected high scoring patches would be returned, as shown in Figure 5. While these patches were able to visually produce patches of images that looked very representative of their class texture, I was unable to use this to help increase classification accuracy.

While my original plan was to run my code against the entire TinyImageNet, I hit several problems. One issue was that TinyImageNet required a lot of RAM, and I was not able to run the code on my laptop. Eventually, I ported the code over to Terminal, but my code was not optimized enough to run particularly fast, so I could only run it a few times before I ran out of time on Terminal. Because of this, I was unable to tune my code on the entirely of TinyImageNet.

4. Experiment

The data set I am working with has 200 different classes to classify images as. The data is already separated into training, validation and test sets and will be used as such. The data set is very large and has many classes compared to data sets I have worked on previously, so it will provide a relatively good example of how well specific features will work in real life situations.

My original plan for evaluation was that the scores that my algorithm outputs into an Evaluation Server provided by the class in order to see how well the classifier does on scoring. However, I was unable to run my code on the entirety of tiny image net. In the end, I tried to evaluate my code by just scoring it off of TinyImageNet-A or TinyImageNet-B. This was a small enough set that my code would run on my laptop, even though it took some time to run. This meant that I was unable to tune my results significantly.

From the homework, my score using the pretrained models from TinyImageNet-A was about 36% accurate, and my score using fine tuning of the pretrained models on TinyImageNet-B was about 23% accurate. When I included my new features added in right before the final output layer, with the weights for them originally set to 0, my score was still 36% accurate with TinyImageNet-A and about 22% on TinyImageNet-B. Theoretically, With the weights set to 0, the scores should have been identical to the original scores. However, due to the inability to tune it as well, the score on TinyImageNet-B suffered a bit and the score for TinyImageNet-A did not change at all. Interestingly, attempting to fine tune the pretrained model with additional features made it perform worse. This could be because it was much better tuned than how my algorithm could work, and attempting to tune it more merely harmed it.

Interestingly enough, texture did seem to help the scoring of the classes you would expect it to help, though it is hard to tell whether it is luck due to the fact that there was only 50 validation images per class. For example, The validation score for lemons hovered at around 40% accuracy before the texture features, and averaged over 80% after texture. It could also just be that the ”texture” that defined lemons was just a blotch of the color yellow, which would not have shown up as much in many other images, making my classifier just much more sensitive to regions of color.

However, adding in SIFT features did not seem to effect validation scores for individual classes apart from just random variance.

5. Conclusion

While the SIFT and texture features were interesting, I was unable to run them for long enough to properly tune them. Perhaps in the future if I had a stronger laptop or had more free time on terminal, I could have gotten more impressive results.

References

[1] Lorenzo Seidenar, *Local Pyramidal Descriptors for Image Recognition*. IEEE Trans Pattern Anal Mach Intell. 2013 Nov 22

[2] Robert M Haralick, *Textural Features for Image Classification*. IEEE Transactions on Systems, Man and Cybernetics Vol SMC-3 No 6 pp. 610-621 1973.

[3] David G. Lowe, *Distinctive Image Features from Scale-Invariant Keypoints*. The International Journal of Computer Vision, 2004.

[4] Bradski, G., *OpenCV*. Dr. Dobb's Journal of Software Tools, 2000

6. Supplementary Materials

```
import numpy as np
import scipy.signal as signal

def texture_features(X,F=2,HH=10,WW=10\
,iterations=10,compares=2,threshold=.5):
"""
Computer experimental texture features
Inputs:
- X: Input data of shape (C, H, W)
- F: Number of textures to find
Outputs:
- textures: (F, C, HH, WW) array of F
different textures detected, of height
and width HHxWW
"""
N,C,H,W=X.shape
textures=np.zeros([2*F,C,HH,WW])
texture_scores=[-1]*(2*F)
for iter in xrange(iterations):
for i in xrange(2*F):
if texture_scores[i]<threshold:
xx=np.random.randint(H-HH)
yy=np.random.randint(W-WW)
textures[i]=X[:,xx:(xx+HH),\
yy:(yy+WW)]
texture_scores[i]=0
for x in xrange(H-HH):
for y in xrange(W-WW):
texture_scores[i]=np.max(\
texture_scores[i],np.correlate(\
X[:,x:(x+HH),y:(y+WW)].flatten(),\
textures[i].flatten()))
index=np.argsort(texture_scores)
texture_scores=np.array(texture_scores)\
[index[::-1]]
textures=np.array(textures)[index[::-1]]
texture_scores[F:2*F]=-1
return texture_scores[0:F],textures[0:F]

def find_textures(X,F=10,HH=10,WW=10,\
iterations=10,compares=2,threshold=4e6):
"""
Computer experimental texture features

Inputs:
- X: Input data of shape (N, C, H, W)
OF THE SAME CLASS
- F: Number of textures to find PER CLASS

Outputs:
- textures: (F, C, HH, WW) array of F
different textures detected, of
height and width HHxWW
"""
N,C,H,W=X.shape
textures=np.zeros([2*F,C,HH,WW])
texture_scores=[-1]*(2*F)
for iter in xrange(iterations):
for i in xrange(2*F):
if texture_scores[i]==-1:
zz=np.random.randint(N)
xx=np.random.randint(H-HH)
yy=np.random.randint(W-WW)
textures[i]=X[zz,:,xx:(xx+HH),yy:(yy+WW)]
z=np.random.randint(N,size=compares)
texture_scores[i]=0
for x in xrange(H-HH):
for y in xrange(W-WW):
#was originally corr
if np.max(np.sum((X[z,:,x:(x+HH),y:(y+WW)]\
-textures[i])**2,axis=(1,2,3)))<threshold:
texture_scores[i]+=1
else:
z=np.random.randint(N,size=compares)
for x in xrange(H-HH):
for y in xrange(W-WW):
#was originally corr
if np.max(np.sum((X[z,:,x:(x+HH),y:(y+WW)]\
-textures[i])**2,axis=(1,2,3)))<threshold:
texture_scores[i]+=1
texture_scores[i]+=1
texture_scores[i]/=2
```

```
index=np.argsort(texture_scores)
texture_scores=np.array(texture_scores)\
[index[::-1]]
textures=np.array(textures)[index[::-1]]
texture_scores[F:2*F]=-1
return texture_scores[0:F],textures[0:F]

def score_textures(X,textures,threshold=2e6):
"""
Computer experimental texture features
Inputs:
- X: Input data of shape (N, C, H, W)
- textures: (F, C, HH, WW) array of F different
textures, of height and width HHxWW
Outputs:
- feature_scores: (N,F) array of N different
textures scores for F textures
"""
N,C,H,W=X.shape
F,C,HH,WW=textures.shape
feature_scores=np.zeros([N,F])
for n in xrange(N):
for f in xrange(F):
for x in xrange(H-HH):
for y in xrange(W-WW):
if np.sum((X[n,:,x:(x+HH),y:(y+WW)]\
-textures[f])**2)<threshold:
feature_scores[n,f]+=1
return feature_scores

# Visualize some examples of the training data
from cs231n.classifiers.features import find_textures
from cs231n.classifiers.features import score_textures
classes_to_show = 5
examples_per_class = 10
class_idxs = np.random.choice(len(class_names), \
size=classes_to_show, replace=False)
for i, class_idx in enumerate(class_idxs):
train_idxs, = np.nonzero(y_train == class_idx)
scores,textures= find_textures(X_train[train_idxs.astype(int)]\
+ mean_img,examples_per_class)
print scores
for j, texture in enumerate(textures):
img = texture
img = img.transpose(1, 2, 0).astype('uint8')
plt.subplot(examples_per_class, \
classes_to_show, 1 + i + classes_to_show * j)
if j == 0:
plt.title(class_names[class_idx][0])
plt.imshow(img)
plt.gca().axis('off')
score_textures(X_train,textures)
plt.show()
```